Databricks Spark

知识库 简体中文

Andy Ai

Published with GitBook

Table of Contents

1. Introduction
2. 最佳实践
   i. 避免使用 GroupByKey
   ii. 不要将大型 RDD 的所有元素拷贝到请求驱动者
3. 常规故障处理
   i. Job aborted due to stage failure: Task not serializable
   ii. 缺失依赖
   iii. 执行 start-all.sh 错误 - Connection refused
   iv. Spark 组件之间的网络连接问题
4. 性能 & 优化
   i. 一个 RDD 有多少个分区
   ii. 数据本地性
5. Spark Streaming
   i. ERROR OneForOneStrategy

Databricks Spark 知识库

- 最佳实践
  - 避免使用 GroupByKey
  - 不要将大型 RDD 的所有元素拷贝到请求驱动者
- 常规故障处理
  - Job aborted due to stage failure: Task not serializable
  - 缺失依赖
  - 执行 start-all.sh 错误 - Connection refused
  - Spark 组件之间的网络连接问题
- 性能 & 优化
  - 一个 RDD 有多少个分区
  - 数据本地性
- Spark Streaming
  - ERROR OneForOneStrategy

Copyright

本文翻译自: http://databricks.gitbooks.io/databricks-spark-knowledge-base/ 著作权归原作者所有。

License

此内容使用的授权许可请查看这里。

最佳实践

- 避免使用 GroupByKey
- 勿在大型 RDD 上直接调用 collect

避免使用 GroupByKey

让我们看一下使用两种不同的方式去计算单词的个数，第一种方式使用 `reduceByKey` 另外一种方式使用 `groupByKey` ：

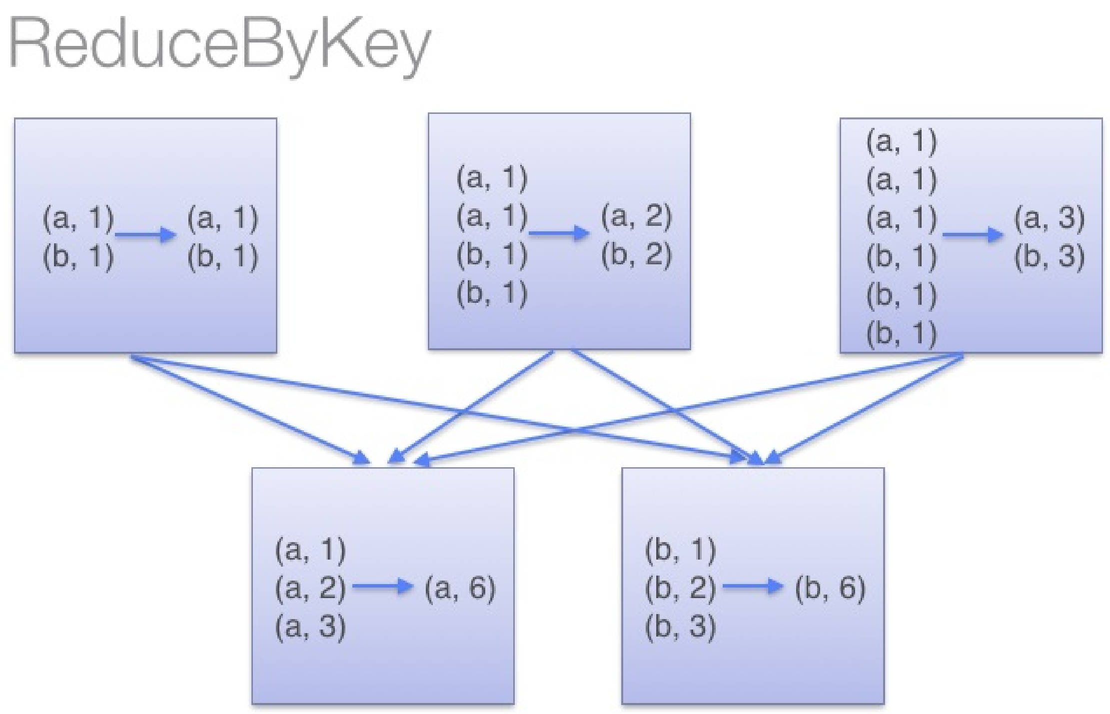
```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

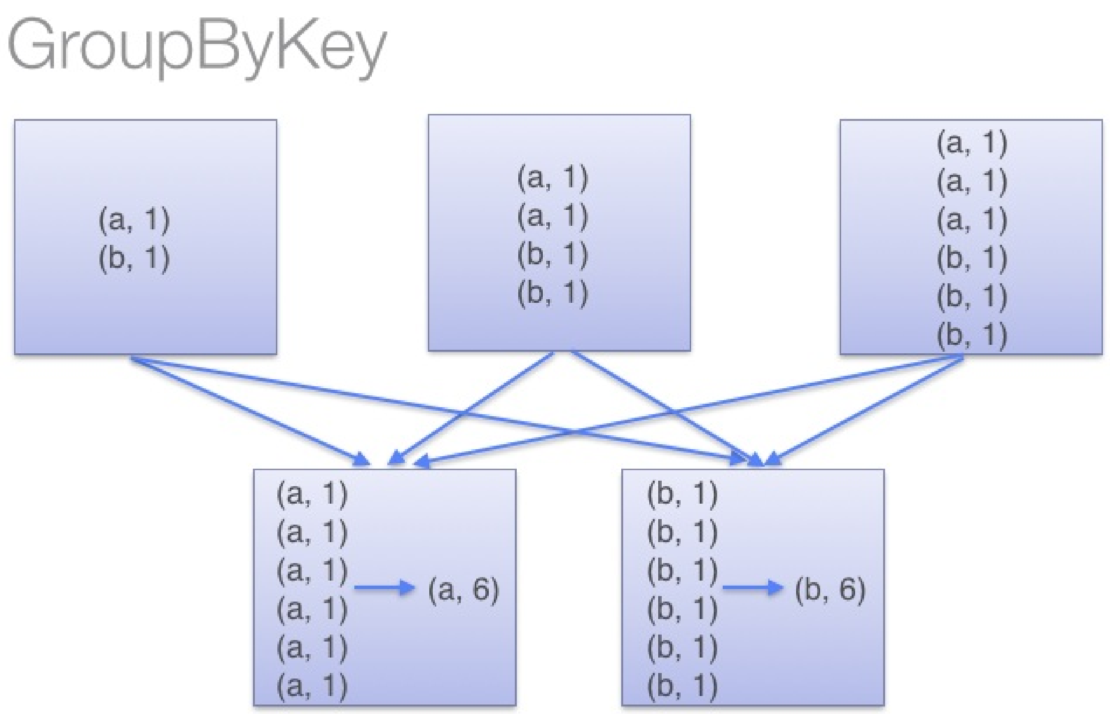
虽然两个函数都能得出正确的结果， `reduceByKey` 更适合使用在大数据集上。 这是因为 Spark 知道它可以在每个分区移动数据之前将输出数据与一个共用的 key 结合。

借助下图可以理解在 `reduceByKey` 里发生了什么。 注意在数据对被搬移前同一机器上同样的 key 是怎样被组合的(`reduceByKey` 中的 lamdba 函数)。然后 lamdba 函数在每个区上被再次调用来将所有值 reduce 成一个最终结果。



另一方面，当调用 `groupByKey` 时，所有的键值对(key-value pair) 都会被移动。在网络上传输这些数据非常没有必要。

为了确定将数据对移到哪个主机，Spark 会对数据对的 key 调用一个分区算法。 当移动的数据量大于单台执行机器内存总量时 Spark 会把数据保存到磁盘上。 不过在保存时每次会处理一个 key 的数据，所以当单个 key 的键值对超过内存容量会存在内存溢出的异常。 这将会在之后发行的 Spark 版本中更加优雅地处理，这样的工作还可以继续完善。 尽管如此，仍应避免将数据保存到磁盘上，这会严重影响性能。



你可以想象一个非常大的数据集，在使用 `reduceByKey` 和 `groupByKey` 时他们的差别会被放大更多倍。

以下函数应该优先于 `groupByKey` ：

- `combineByKey` 组合数据，但是组合之后的数据类型与输入时值的类型不一样。
- `foldByKey` 合并每一个 key 的所有值，在级联函数和“零值”中使用。

阅读原文

不要将大型 RDD 的所有元素拷贝到请求驱动者

如果你的驱动机器(submit 请求的机器)内存容量不能容纳一个大型 RDD 里面的所有数据，不要做以下操作：

```
val values = myVeryLargeRDD.collect()
```

Collect 操作会试图将 RDD 里面的每一条数据复制到驱动机器(submit 请求的机器)上，这时候会发生内存溢出和崩溃。

相反，你可以调用 `take` 或者 `takeSample` 来确保数据大小的上限。或者在你的 RDD 中使用过滤或抽样。

同样，要谨慎使用下面的操作，除非你能确保数据集小到足以存储在内存中：

- `countByKey`
- `countByValue`
- `collectAsMap`

如果你确实需要将 RDD 里面的大量数据保存在内存中，你可以将 RDD 写成一个文件或者把 RDD 导出到一个容量足够大的数据库中。

阅读原文

常规故障处理

- Job aborted due to stage failure: Task not serializable
- 缺失依赖
- 执行 start-all.sh 错误 - Connection refused
- Spark 组件之间的网络连接问题

Job aborted due to stage failure: Task not serializable:

如果你能看到以下错误：

```
org.apache.spark.SparkException: Job aborted due to stage failure: Task not serializable:
```

上述的错误在这个时候会被触发：当你在 master 上初始化一个变量，但是试图在 worker 上使用。在这个示例中， Spark Streaming 试图将对象序列化之后发送到 worker 上，如果这个对象不能被序列化就会失败。思考下面的代码片段：

```
NotSerializable notSerializable = new NotSerializable();
JavaRDD<String> rdd = sc.textFile("/tmp/myfile");

rdd.map(s -> notSerializable.doSomething(s)).collect();
```

这段代码会触发那个错误。这里有一些建议修复这个错误：

- 让 class 实现序列化
- 在作为参数传递给 map 方法的 lambda 表达式内部声明实例
- 在每一台机器上创建一个 NotSerializable 的静态实例
- 调用 `rdd.forEachPartition` 并且像下面这样创建 NotSerializable 对象：

```
rdd.forEachPartition(iter -> {
  NotSerializable notSerializable = new NotSerializable();

  // ...Now process iter
});
```

阅读原文

缺失依赖

在默认状态下，Maven 在 build 的时候不会包含所依赖的 jar 包。当运行一个 Spark 任务，如果 Spark worker 机器上没有包含所依赖的 jar 包会发生类无法找到的错误(`ClassNotFoundException`)。

有一个简单的方式，在 Maven 打包的时候创建 *shaded* 或 *uber* 任务可以让那些依赖的 jar 包很好地打包进去。

使用 `<scope>provided</scope>` 可以排除那些没有必要打包进去的依赖，对 Spark 的依赖必须使用 `provided` 标记，因为这些依赖已经包含在 Spark cluster中。在你的 worker 机器上已经安装的 jar 包你同样需要排除掉它们。

下面是一个 Maven pom.xml 的例子，工程了包含了一些需要的依赖，但是 Spark 的 libraries 不会被打包进去，因为它使用了 `provided` ：

```
<project>
    <groupId>com.databricks.apps.logs</groupId>
    <artifactId>log-analyzer</artifactId>
    <modelVersion>4.0.0</modelVersion>
    <name>Databricks Spark Logs Analyzer</name>
    <packaging>jar</packaging>
    <version>1.0</version>
    <repositories>
        <repository>
            <id>Akka repository</id>
            <url>http://repo.akka.io/releases</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency> <!-- Spark -->
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.10</artifactId>
            <version>1.1.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency> <!-- Spark SQL -->
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.10</artifactId>
            <version>1.1.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency> <!-- Spark Streaming -->
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.10</artifactId>
            <version>1.1.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency> <!-- Command Line Parsing -->
            <groupId>commons-cli</groupId>
            <artifactId>commons-cli</artifactId>
            <version>1.2</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
```

```
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>2.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*.SF</exclude>
                                <exclude>META-INF/*.DSA</exclude>
                                <exclude>META-INF/*.RSA</exclude>
                            </excludes>
                        </filter>
                    </filters>
                    <finalName>uber-${project.artifactId}-${project.version}</finalName>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

阅读原文

执行 start-all.sh 错误: Connection refused

如果是使用 Mac 操作系统运行 start-all.sh 发生下面错误时：

```
% sh start-all.sh
starting org.apache.spark.deploy.master.Master, logging to ...
localhost: ssh: connect to host localhost port 22: Connection refused
```

你需要在你的电脑上打开 “远程登录” 功能。进入 `系统偏好设置` ---> `共享` 勾选打开 `远程登录`。

阅读原文

置。

Spark 组件之间的网络连接问题

Spark 组件之间的网络连接问题会导致各式各样的警告/错误：

- **SparkContext <-> Spark Standalone Master:**

  如果 SparkContext 不能连接到 Spark standalone master，会显示下面的错误

  ```
  ERROR AppClient$ClientActor: All masters are unresponsive! Giving up.
  ERROR SparkDeploySchedulerBackend: Spark cluster looks dead, giving up.
  ERROR TaskSchedulerImpl: Exiting due to error from cluster scheduler: Spark cluster
  ```

  如果 driver 能够连接到 master 但是 master 不能回连到 driver上，这时 Master 的日志会记录多次尝试连接 driver 失败并且会报告不能连接：

  ```
  INFO Master: Registering app SparkPi
  INFO Master: Registered app SparkPi with ID app-XXX-0000
  INFO: Master: Removing app app-app-XXX-0000
  [...]
  INFO Master: Registering app SparkPi
  INFO Master: Registered app SparkPi with ID app-YYY-0000
  INFO: Master: Removing app app-YYY-0000
  [...]
  ```

  在这样的情况下，master 报告应用已经被成功地注册了。但是注册成功的通知 driver 接收失败了， 这时 driver 会自动尝试几次重新连接直到失败的次数太多而放弃重试。其结果是 Master web UI 会报告多个失败的应用，即使只有一个 SparkContext 被创建。

建议

如果你遇到上述的任何错误：

- 检查 workers 和 drivers 配置的 Spark master 的地址就是在 Spark master web UI/日志中列出的那个地址。
- 设置 driver，master，worker 的 `SPARK_LOCAL_IP` 为集群的可寻地址主机名。

配置 hostname/port

这节将描述我们如何绑定 Spark 组件的网络接口和端口。

在每节里，配置会按照优先级降序的方式排列。如果前面所有配置没有提供则使用最后一条作为默认配置。

SparkContext actor system:

Hostname:

- `spark.driver.host` 属性

- 如果 `SPARK_LOCAL_IP` 环境变量的设置是主机名(hostname)，就会使用设置时的主机名。如果 `SPARK_LOCAL_IP` 设置的是一个 IP 地址，这个 IP 地址会被解析为主机名。
- 使用默认的 IP 地址，这个 IP 地址是Java 接口 `InetAddress.getLocalHost` 方法的返回值。

Port:

- `spark.driver.port` 属性。
- 从操作系统(OS)选择一个临时端口。

Spark Standalone Master / Worker actor systems:

Hostname:

- 当 `Master` 或 `Worker` 进程启动时使用 `--host` 或 `-h` 选项(或是过期的选项 `--ip` 或 `-i`)。
- `SPARK_MASTER_HOST` 环境变量(仅应用在 `Master` 上)。
- 如果 `SPARK_LOCAL_IP` 环境变量的设置是主机名(hostname)，就会使用设置时的主机名。如果 `SPARK_LOCAL_IP` 设置的是一个 IP 地址，这个 IP 地址会被解析为主机名。
- 使用默认的 IP 地址，这个 IP 地址是Java 接口 `InetAddress.getLocalHost` 方法的返回值。

Port:

- 当 `Master` 或 `Worker` 进程启动时使用 `--port` 或 `-p` 选项。
- `SPARK_MASTER_PORT` 或 `SPARK_WORKER_PORT` 环境变量(分别应用到 `Master` 和 `Worker` 上)。
- 从操作系统(OS)选择一个临时端口。

阅读原文

性能优化

- 一个 RDD 有多少个分区
- 数据本地性

一个 RDD 有多少个分区？

在调试和故障处理的时候，我们通常有必要知道 RDD 有多少个分区。这里有几个方法可以找到这些信息：

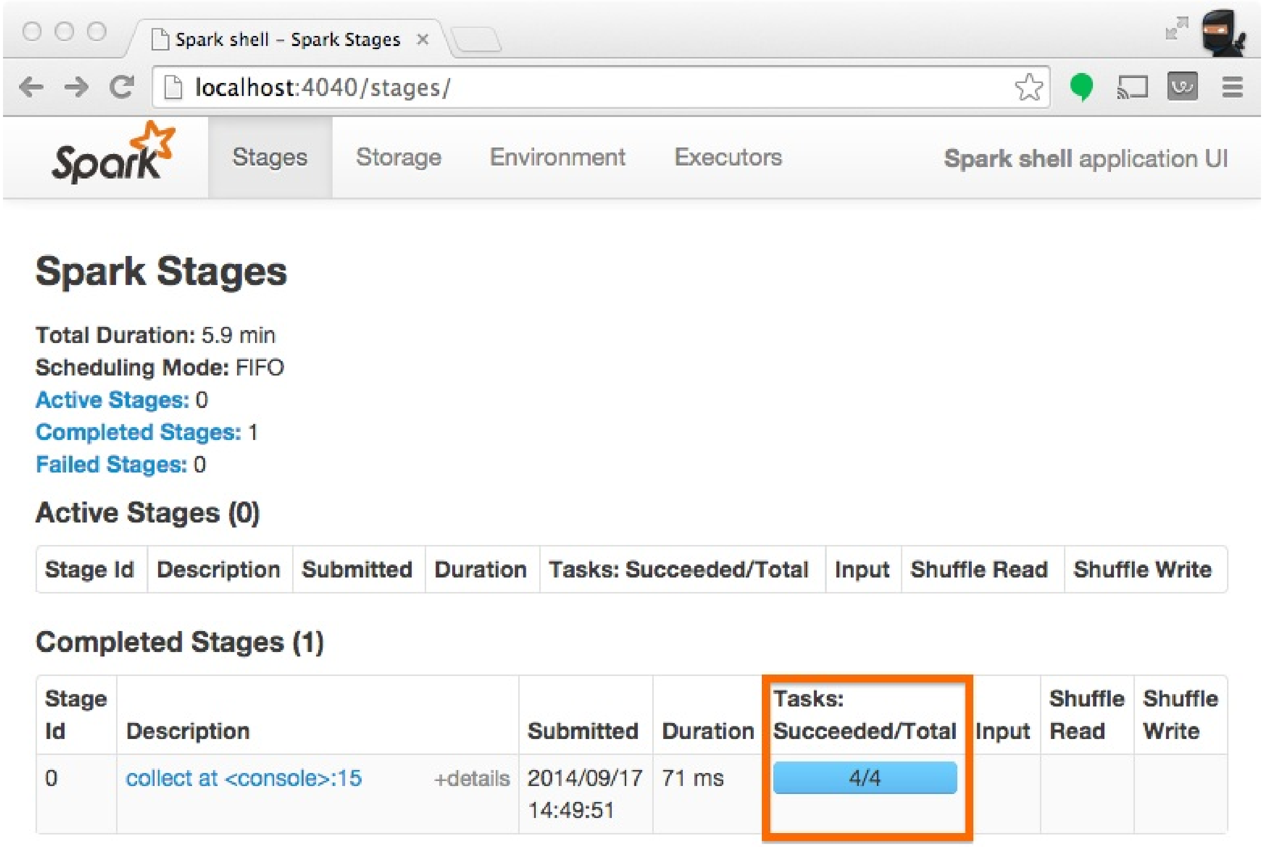
使用 UI 查看在分区上执行的任务数

当 stage 执行的时候，你可以在 Spark UI 上看到这个 stage 上的分区数。 下面的例子中的简单任务在 4 个分区上创建了共 100 个元素的 RDD ，然后在这些元素被收集到 driver 之前分发一个 map 任务：

```
scala> val someRDD = sc.parallelize(1 to 100, 4)
someRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <cons

scala> someRDD.map(x => x).collect
res1: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
```

在 Spark 的应用 UI 里，从下面截图上看到的 "Total Tasks" 代表了分区数。



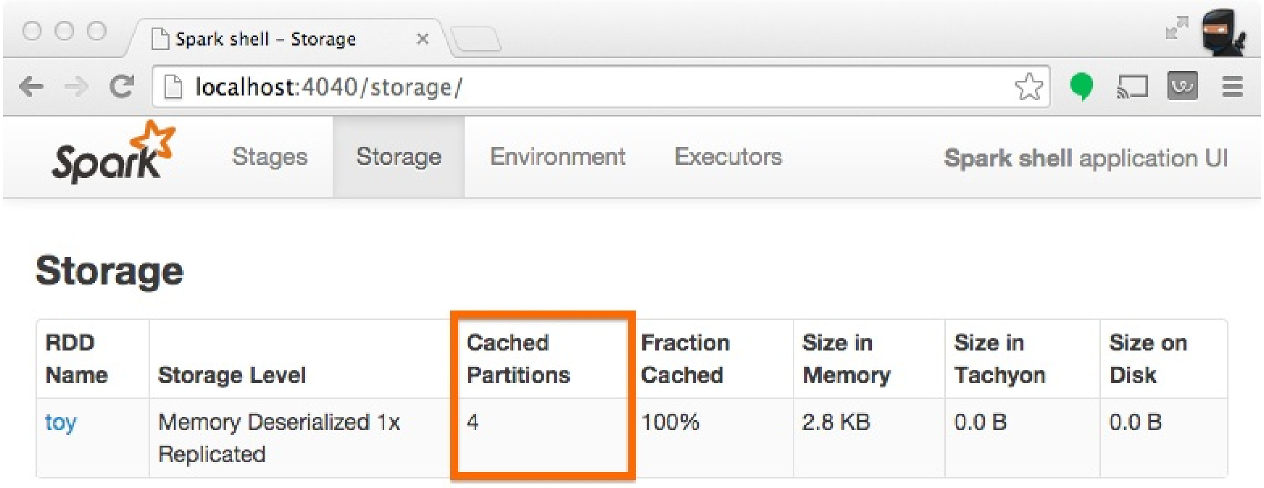
使用 UI 查看分区缓存

持久化(即缓存) RDD 时通常需要知道有多少个分区被存储。 下面的这个例子和之前的一样，除了现在我们要对 RDD 做缓存处理。操作完成之后，我们可以在 UI 上看到这个操作导致什么被我们存储了。

```
scala> someRDD.setName("toy").cache
res2: someRDD.type = toy ParallelCollectionRDD[0] at parallelize at <console>:12
```

```
scala> someRDD.map(x => x).collect
res3: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
```

注意：下面的截图有 4 个分区被缓存。



RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
toy	Memory Deserialized 1x Replicated	4	100%	2.8 KB	0.0 B	0.0 B

编程查看 RDD 分区

在 Scala API 里，RDD 持有一个分区数组的引用， 你可以使用它找到有多少个分区：

```
scala> val someRDD = sc.parallelize(1 to 100, 30)
someRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <cons

scala> someRDD.partitions.size
res0: Int = 30
```

在 Python API 里, 有一个方法可以明确地列出有多少个分区：

```
In [1]: someRDD = sc.parallelize(range(101),30)

In [2]: someRDD.getNumPartitions()
Out[2]: 30
```

注意：上面的例子中，是故意把分区的数量初始化成 30 的。

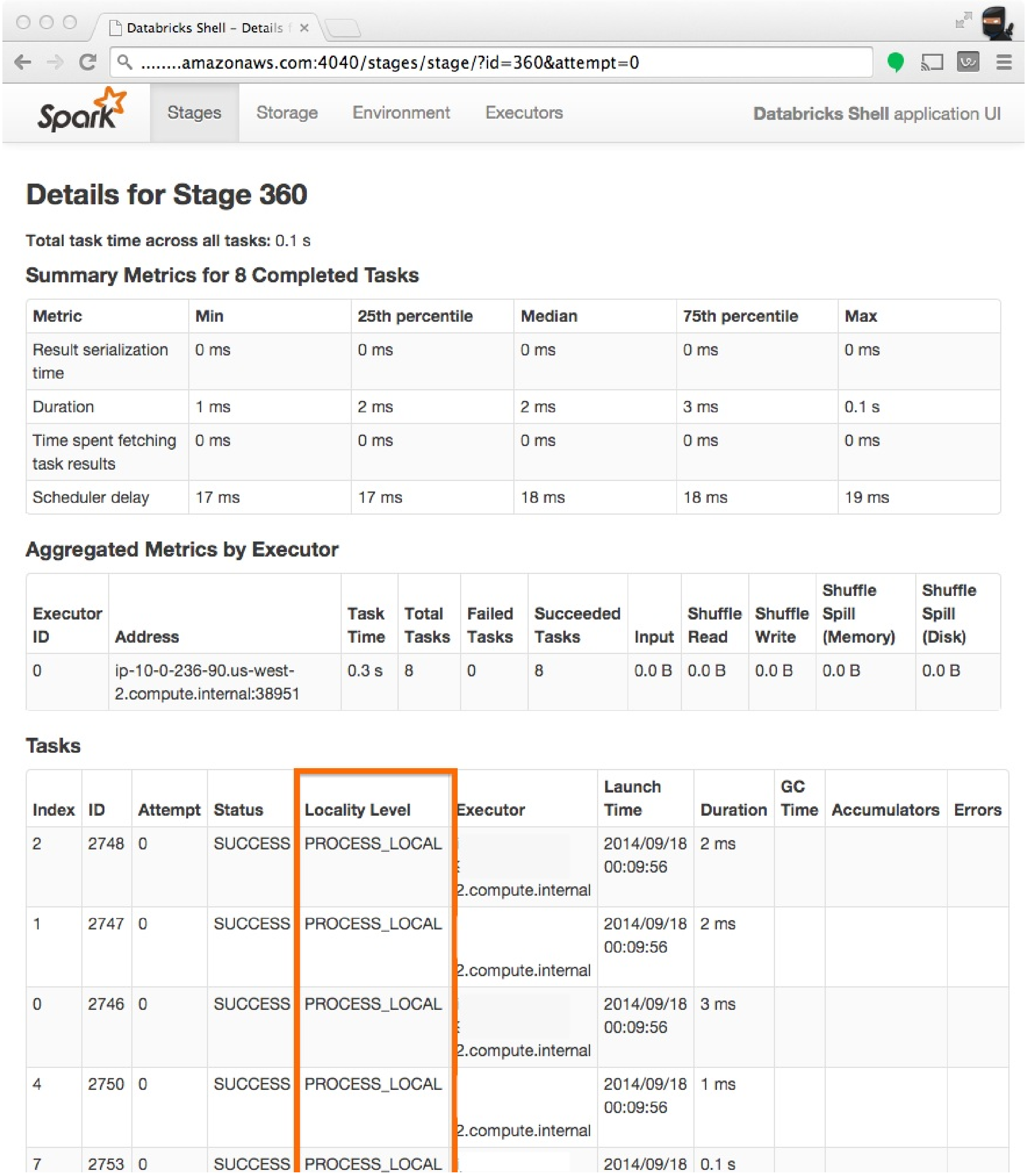
阅读原文

数据本地性

Spark 是一个并行数据处理框架，这意味着任务应该在离数据尽可能近的地方执行(既 最少的数据传输)。

检查本地性

检查任务是否在本地运行的最好方式是在 Spark UI 上查看 stage 信息，注意下面截图中的 "Locality Level" 列显示任务运行在哪个地方。



Details for Stage 360

Total task time across all tasks: 0.1 s

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	0 ms
Duration	1 ms	2 ms	2 ms	3 ms	0.1 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	17 ms	17 ms	18 ms	18 ms	19 ms

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	ip-10-0-236-90.us-west-2.compute.internal:38951	0.3 s	8	0	8	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Tasks

Index	ID	Attempt	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Accumulators	Errors
2	2748	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	2 ms			
1	2747	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	2 ms			
0	2746	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	3 ms			
4	2750	0	SUCCESS	PROCESS_LOCAL	2.compute.internal	2014/09/18 00:09:56	1 ms			
7	2753	0	SUCCESS	PROCESS_LOCAL		2014/09/18	0.1 s			

调整本地性配置

你可以调整 Spark 在每个数据本地性阶段(data local --> process local --> node local --> rack local --> Any)上等待的时长。更多详细的参数信息请查看程序配置文档的 Scheduling 章节里类似于 `spark.locality.*` 的配置。

阅读原文

Spark Streaming

- ERROR OneForOneStrategy

ERROR OneForOneStrategy

如果你在 Spark Streaming 里启用 checkpointing，forEachRDD 函数使用的对象都应该可以被序列化(Serializable)。否则会出现这样的异常 "ERROR OneForOneStrategy: ... java.io.NotSerializableException:"

```
JavaStreamingContext jssc = new JavaStreamingContext(sc, INTERVAL);

// This enables checkpointing.
jssc.checkpoint("/tmp/checkpoint_test");

JavaDStream<String> dStream = jssc.socketTextStream("localhost", 9999);

NotSerializable notSerializable = new NotSerializable();
dStream.foreachRDD(rdd -> {
      if (rdd.count() == 0) {
        return null;
      }
      String first = rdd.first();

      notSerializable.doSomething(first);
      return null;
    }
);

// This does not work!!!!
```

按照下面的方式之一进行修改，上面的代码才能正常运行：

- 在配置文件里面删除 `jssc.checkpoint` 这一行关闭 checkpointing。
- 让对象能被序列化。
- 在 forEachRDD 函数里面声明 NotSerializable，下面的示例代码是可以正常运行的：

```
JavaStreamingContext jssc = new JavaStreamingContext(sc, INTERVAL);

jssc.checkpoint("/tmp/checkpoint_test");

JavaDStream<String> dStream = jssc.socketTextStream("localhost", 9999);

dStream.foreachRDD(rdd -> {
      if (rdd.count() == 0) {
        return null;
      }
      String first = rdd.first();
      NotSerializable notSerializable = new NotSerializable();
      notSerializable.doSomething(first);
      return null;
    }
);

// This code snippet is fine since the NotSerializable object
// is declared and only used within the forEachRDD function.
```

阅读原文